

Designing and Simulating a Virtual Compiler Design Laboratory

Sayantani Saha _12019002004082,
Saheli Sau_ 12019002004104,
Riyanka choudhary_12019002004095,
Saranya Ghosh_12019002004091,
Saswata Das_12019002004100,

Dept. of Information Technology, Institute of Engineering and Management, Kolkata
May 10 ,2022

Abstract

In this post-pandemic era, the importance of online education and its benefits are not unknown to us. From distance education to self-paced learning, it is the virtual education that has helped to learn the most out of any particular subject. Keeping these advantages in mind, we have developed a website using HTML5, CSS3 and Javascript in the frontend to simulate, visualize and ease the learning process of Compiler Designing. To learn the core concepts related to the working of a compiler, it is an utmost necessity to grasp the phases of Compiler Design thoroughly. Our website is equipped to explain and simulate five different compiler designing related tasks to ensure proper understanding and enhance the interactivity in the learner-learning process.

<https://gr13vlabiem.netlify.app/>

Introduction

Compilers translate high-level programming languages such as C and C++ into assembly code for a target processor. Used for decades to program desktop operating systems and applications, compilers are among the most widespread software tools. According to definition, a compiler consists of a series of phases that sequentially analyze given forms of a program, and synthesize new ones, beginning with the sequence of characters constituting a source program to be compiled and ultimately producing a relocatable object module that can be linked with others and loaded into a machine's memory to be executed. In

the whole Compiler Designing process, namely 4 stages are involved in the frontend (Lex-ical Analysis, Syntax Analysis, Semantic Analysis, Intermediate Code Generation) and 2 optional but recommended stages are involved in the backend (Code Optimization, Code Generation). Early in the compilation process the source program appears as a stream of characters. The two subprocesses of "scanning" and "screening" constitute the process known as lexical analysis. Scanning involves finding substrings of characters that constitute units called textual elements (Words, Punctuation, Single and Multi-Character Operators etc.). Screening involves discarding some textual elements, such as spaces and comments, and the recognition of reserved symbols, such as the key words and operators, used in the particular language being translated. It is the output of this process, usually called a token stream, that is the input to the parser. After Lexical Analysis, the syntactical structure of the given input is checked in the Syntax Analysis Phase. Through generating a parse tree, it checks whether the structure is valid or not. The task of Semantic Analysis is to determine properties and check conditions that are relevant for the well-formedness of the programs according to the rules of programming language, but that can go beyond of Context-free Grammars.

The next step involves Intermediate Code Generation, which deals with the breaking down of the input code into smaller fragments in order to execute and handle the code more efficiently. In the last two phases in the backend, the high-level code gets translated into low-level machine code with some optimizations being performed to the code.

Related Works

Previously, the concepts of Compiler Design was implemented in the field of distributed quantum computing. To establish the network and communication functionalities provided by the Quantum Internet through remote quantum processing units(QPUs), designing of a proper compiler is an inseparable task . Also in India, several attempts were taken to ease the process of learning a difficult concept like compiler design in various educational institutes such as Delhi University . Institutes like IIT Bombay has also developed a Virtual Simulation Laboratory(V-Lab) to deliver practical, real-life and hands on concepts regarding various Compiler Design Concepts(CDC). Also, there have been major efforts in the field of CDC regarding compiler support for distributed memory parallel machines.

Language used

- Hypertext Markup Language 5 (For Building the skeleton of the website)
- Cascading Style Sheets 3 (For stylization and beautification)
- Javascript (For creating user-interactive panel and applying animations and effects)

Website Construction and Design

We have constructed our website keeping an user-friendly and virtual education-oriented approach in mind. We have focused on five different core CDC topics:

- Comment Checking
- Lexical Analysis
- Identifier Validation
- Left Right Derivation
- Parse Tree Generator

For the ease of navigation and improved interactivity, every topic contains five segments beside the simulation process:

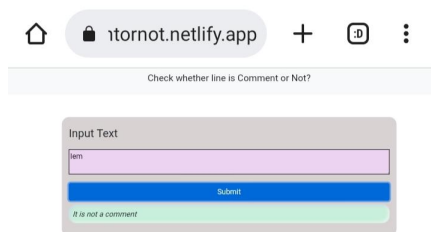
- Aim of the Lab
- Theory behind the key concepts
- Input/Output(I/O) Examples
- Discussion
- Source of error and error recovery

1. Comment checking

Comments are a crucial part of a code as they act as a necessary tool for understanding the different segments of code. It helps to describe, analyze and keep a track of important takeaways in a code snippet. But, for a compiler they act as a redundant part as they don't contribute anything to the execution of the program itself and hence should be removed with immediate effect. This is where identifying comments become a mandate. In the simulation segment, the web application takes an input strings and determines if it is a comment or not. It does so by checking the string itself. On the basis of identification, they can broadly be classified into two categories:

- Single Line Comment(If the string starts with '//')
- Multi Line Comment(If the String starts with '/' and ends with '/')

To visualize the actual simulation process for the Comment Checking, go to the link provided



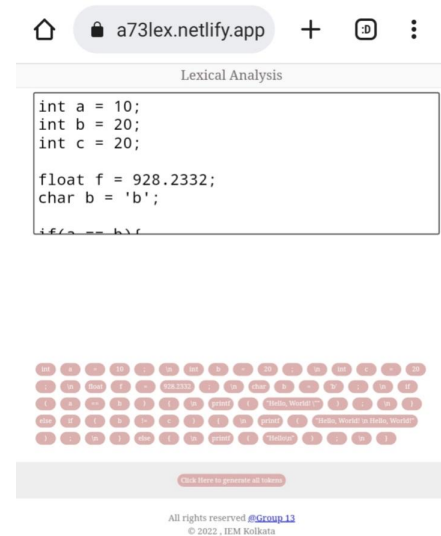
Simulation of comment checking

Lexical Analysis

It's the first phase of Compiler Design which mainly deals with classifying tokens from a given code snippet. Tokens can be considered as the smallest logical units of a program, which are generally identified by the longest-match rule in a lexical analyzer. In our simulator, we filter out the tokens from a given input stream of characters (Strings), which consists of:

- Keywords
- Function Names
- Identifiers
- String Constants
- Special Characters
- Operators

To visualize the actual simulation process for the Lexical Analyzer, go to the link provided



Identifying tokens

Besides selecting the tokens, it also eliminates the non-tokens from the program, such as:

- Tabs
- Whitespaces
- Comment

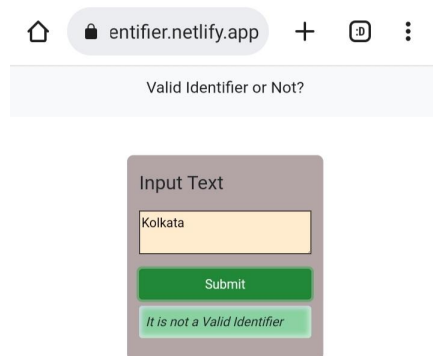
Validating Identifier

Identifiers are the combinations of alphanumeric characters that are eligible for using as an

user-defined name for any variable, function, class, method etc. So, validating an identifier is an important task for any compiler. Our simulator also checks whether a given string is a valid identifier or not by checking some pre-defined parameters for qualification, like:

- Starting with an underscore()
- Starting with any lowercase (a-z) or uppercase(A-Z) letters
- Absence of any whitespace
- Not having any special characters(#,%, \$ etc.) after the first character
- Shorter than 31 characters, in length.

To visualize the actual simulation process for the Identifier Validation, go to the link provided

The image shows a web browser window with the address bar displaying 'entifier.netlify.app'. Below the address bar, there is a header area with the text 'Valid Identifier or Not?'. The main content area contains a form with a label 'Input Text' above a text input field. The input field contains the text 'Kolkata'. Below the input field is a green 'Submit' button. At the bottom of the form, there is a green message box that says 'It is not a Valid Identifier'.

Simulation of validating Identifier

Left Right Derivation

Context free Grammar(CFG) comes under type-2 grammar which is used to generate all the possible combinations of a string written in a formal language. So, CFG is beneficial in the domain of compiler design as well as parser programs. CFGs are used to derive a parse tree which is used to represent the semantic information of a given input string. In our simulator, we are intended to display the leftmost and the rightmost derivation of a given set of CFG.

Derivation tree gives an overview of how each variable is substituted in the process from the root node to the leaf nodes[8]. In case of leftmost and rightmost derivations, the string is derived by expansion of the leftmost and rightmost non-terminals respectively, at each step. To visualize the actual simulation process for the Left-Right Derivation, go to the link provided here.

Production rules:

$E \rightarrow E + E$
 $E \rightarrow E * E$
 $E \rightarrow a | b$

Input

$a * b + a$

Submit

The leftmost derivation is:

$E \rightarrow E + E$
 $E \rightarrow E * E$
 $E \rightarrow a * E$
 $E \rightarrow a * b * E$
 $E \rightarrow a * b * a$

The rightmost derivation is:

$E \rightarrow E$
 $E \rightarrow E * E$
 $E \rightarrow E * a$
 $E \rightarrow a * b * a$

All rights reserved @Group 13

© 2022, IEM Kolkata



Simulation of derivation

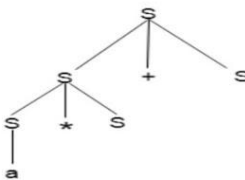
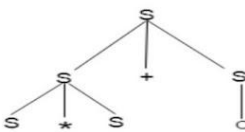
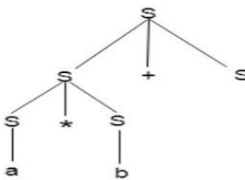
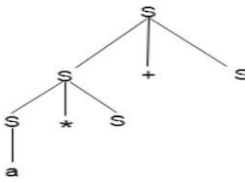
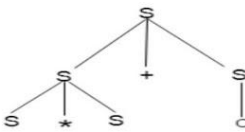
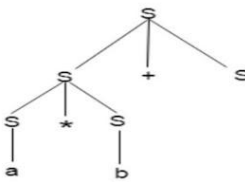
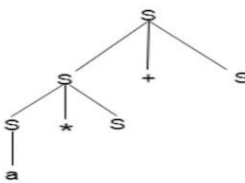
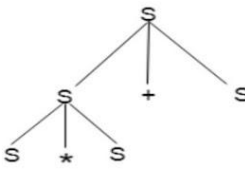
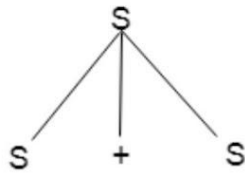
Parse Tree Generator

Consider the example of $a*b+c$

Production rules

```
S → T + T | T  
T → a | b | c
```

Submit



Simulation of parse tree generator

Conclusion

World Wide Web (WWW) has emerged as a leading platform in the recent years for de-livering a plethora of information and quality services. The advent of Web Applications has provided us with an abundance of convenient features like remote accessibility, cross-platform compatibility and rapid development. In this field, JavaScript has contributed a lot in enhancing the overall website aesthetics and boosting up the user experience with improved user interactivity.

In the times of COVID-19 pandemic, when the whole world was behind the closed doors of their home and offline education became unfeasible to carry on, online education was the only way to go. JavaScript helped to facilitate online education by developing E-academic based collaborative learning platform with certain mathematics-based tools such as Geogebra. Also, in as a replacement of traditional laboratories, Virtual labs and Remote labs are being developed using JavaScript for learning the concepts virtually from any corner in the world. It also solves the problem of supervising an entire classroom and tracking and monitoring the progress of the students gets much easier as well. Thus, Virtual Labs play a vital role in revolutionizing the entire educational framework of the globe.

Acknowledgement

First and foremost, We would like to thank Prof. Subhabrata Sengupta (Department of Information Technology, Institute of Engineering and Management) and Prof. Pulak Baral (Department of Information Technology, Institute of Engineering and Management) for guiding and motivating us through the entire timeline of the project. Without their utmost care, advice and support, it would have been impossible to complete this project successfully. Also, we would like to thank all of our teachers, classmates and friends for helping and guiding in the times of difficulty. We can't express our gratitude to everyone.

Limitations

Due to unavailability of resources, input-based simulator was not implemented in the parse tree and left right derivation sections. Instead, a predefined expression was taken into consideration for showing the workings of those sections. Some more core concepts regarding Compiler Design can also be implemented at the later stages of development of the Web application.

References

- [1] JW BACKER et al. An extension of context-free grammars. *Journal of Computer and System Sciences* 15: 4, 647-671, 1968.
- [2] Xuemin Chen, Gangbing Song, and Yongpeng Zhang. Virtual and remote laboratory development: A review. *Earth and Space 2010: Engineering, Science, Construction, and Operations in Challenging Environments*, pages 3843–3852, 2010.
- [3] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of parallel and distributed computing*, 22(3):462–478, 1994.
- [4] Franklin L DeRemer. Lexical analysis. In *Compiler Construction*, pages 109–120. Springer, 1976.
- [5] Davide Ferrari, Angela Sara Cacciapuoti, Michele Amoretti, and Marcello Caleffi. Compiler design for distributed quantum computing. *arXiv preprint arXiv:2012.09680*, 2020.
- [6] Geoffrey C Fox, I Seema, Ken Hirani, Charles Koelbel, Uli Kremer, Chau-wen Tseng, and Min-you Wu. Fortran D language specification. Technical report, Technical Report Rice COMP TR90-141, Department of Computer Science, Rice University, 1990.
- [7] Seymour Ginsburg and Nancy Lynch. Derivation complexity in context-free grammar forms. *SIAM Journal on Computing*, 6(1):123–138, 1977.
- [8] Sandra E Hutchins. Moments of string and derivation lengths of stochastic context-free grammars. *Information Sciences*, 4(2):179–191, 1972.

- [9] Divya Kundra and Ashish Sureka. An experience report on teaching compiler design concepts using case-based and project-based learning approaches. In 2016 IEEE Eighth International Conference on Technology for Education (T4E), pages 216–219. IEEE, 2016.
- [10] Rainer Leupers. Compiler design issues for embedded processors. *IEEE Design & Test of Computers*, 19(4):51–58, 2002.
- [11] Xiaowei Li and Yuan Xue. A survey on server-side approaches to securing web applications. *ACM Computing Surveys (CSUR)*, 46(4):1–29, 2014.
- [12] Steven Muchnick et al. *Advanced compiler design implementation*. Morgan kaufmann, 1997.
- [13] Thomas Ruprecht and Tobias Denking. Implementation of a chomsky-schützenberger-best parser for weighted multiple context-free grammars. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 178–191, 2019.
- [14] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. Quality analysis of source code comments. In *2013 21st international conference on program comprehension (icpc)*, pages 83–92. IEEE, 2013.
- [15] Reinhard Wilhelm, Helmut Seidl, and Sebastian Hack. *Compiler design: syntactic and semantic analysis*. Springer Science & Business Media, 2013.
- [16] Wu Yang, Chey-Woei Tsay, and Jien-Tsai Chan. On the applicability of the longest-match rule in lexical analysis. *Computer Languages, Systems & Structures*, 28(3):273–288, 2002.